Single Restart with Time Stamps for Computational Offloading in a Semi-Online Setting

Jaya Prakash Champati and Ben Liang
Department of Electrical and Computer Engineering, University of Toronto
{champati,liang}@comm.utoronto.ca

***Abstract*—We study the problem of scheduling n tasks on $m + m'$ parallel processors, where the processing times on m processors are known while those on the remaining m' processors are not known a priori. This semi-online model is an abstraction of certain heterogeneous computing systems, e.g., with the m known processors representing local CPU cores and the unknown processors representing remote servers with uncertain availability of computing cycles. Our objective is to minimize the makespan of all tasks. We initially focus on the case $m' = 1$ and propose a semi-online algorithm termed Single Restart with Time Stamps (SRTS), which has time complexity $O(n \log n)$. We derive its competitive ratio in comparison with the optimal offline solution. If the unknown processing times are deterministic, the competitive ratio of SRTS is shown to be either always constant or asymptotically constant in practice, respectively in cases where the processing times are independent and dependent on m. A similar result is obtained when the unknown processing times are random. Furthermore, extending the ideas of SRTS, we propose a heuristic algorithm termed SRTS-Multiple (SRTS-M) for the case $m' > 1$. Besides the proven competitive ratios, simulation results further suggest that SRTS and SRTS-M give superior performance on average over randomly generated task processing times, substantially reducing the makespan over the best known alternatives. Interestingly, the performance gain is more significant for task processing times sampled from heavy-tailed distributions.**

I. Introduction

The problem of parallel task processing on multiple processors has wide-ranging applications in information systems. It is essential to contemporary computing and networking applications, due to the prevalence of multi-core CPUs and the availability of auxiliary resources for computational offloading. Existing studies in parallel task processing can be categorized into three types: *offline*, where the processing times of all tasks on all processors are known a priori; *online*, where no processing time is known until after a task has been processed [1]; and *semi-online*, where some processing time information is known. Scheduling decisions proposed in the research literature generally aim to minimize the makespan of the given tasks, system cost in task processing, or a combination of both. Most of such optimization problems are known to be NP-hard and only approximate solutions are available.

In this work, we study the problem of scheduling computing tasks on $m + m'$ parallel processors, where m processors (*known processors*) are identical and the task processing times on them are known a priori, and the task processing times on the remaining m' processors (*unknown processors*) are unknown before the tasks are processed. Under this semi-online setting, we are interested in finding a schedule that minimizes the makespan of n tasks.

The above semi-online scheduling model can be viewed as an abstraction for several important practical systems. The m processors may model parallel CPU cores in a local device (e.g. smartphone, tablet, etc.) or processors in a local computing cluster. The unknown processors may represent computational servers whose help is enlisted by the local device or the local cluster. In particular, in mobile cloud computing systems, the unknown processors may be shared virtual machine instances in a public cloud [2] [3]; in Mobile Edge Computing (MEC), they may be MEC hosts deployed by a cellular base station [4] [5]; and in cyber foraging/opportunistic computing, they may be neighboring mobile devices or cloudlets where the local device offloads its computational tasks [6]–[10].

The scenario of not knowing the processing times on the remote processors arises due to various factors. For example, an MEC host is shared between network service tasks and the offloaded user tasks from the subscribing mobile devices. Thus, only a fraction of the MEC host's CPU cycles may be allocated to the mobile user. Similarly, in opportunistic computing, a neighboring mobile device may not dedicate all of its CPU cycles to the offloaded tasks. Furthermore, there can be uncertain delays associated with offloading and processing the offloaded tasks. The insights developed from our theoretical model can be used to improve computational offloading in these systems.

Most of the previous studies on parallel processing have focused on either the offline or the fully online scenarios. Offline algorithms are clearly not applicable to our problem. Furthermore, even in the simplest offline setting, makespan minimization is known to be NP-hard [11]. On the other hand, if we ignore the partial knowledge of processing times in our problem, we may use existing online algorithms. In the online setting, if all processors are identical, the well-known *List Scheduling* (LS) algorithm has $\left(2 - \frac{1}{m+m'}\right)$ competitive ratio [12]. However, in our case, the unknown processors are not identical to the known ones. In fact, as shown in Section IV-A, LS has *infinite competitive ratio* for our problem. For online scheduling with non-identical processors, Shmoys *et. al.* has proposed an iterative algorithm that achieves $O(\log n)$ compet-

This work has been supported in part by grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

itive ratio [13]. This algorithm can be applied to our problem. However, as shown in [14] and in Section VII, Shmoys' algorithm does not perform well on average.

Therefore, our objective is to develop a semi-online algorithm that effectively utilizes both the known and unknown processors, to provide both a provable competitive ratio and satisfactory average performance. Instead of deterministic scheduling such as LS, we use the approach of *task restarts* similar to [13], where a task scheduled on a processor may be cancelled and re-scheduled possibly on a different processor. Unlike [13], we observe that only one round of restarts is sufficient for our problem. This is similar in design principle to [14], but as explained in Section II, the problem we consider, the proposed algorithm, and the competitive ratio analysis are substantially different from those in [14].

The main contributions of this work are as follows:

- We first show a negative result, that any semi-online algorithm with a pre-determined scheduling order has infinite competitive ratio. This motivates the need for a more effective dynamic scheduling algorithm.
- We first focus on the important case of $m' = 1$, which represents, e.g., the case of mobile cloud computing with m local CPU cores and a more powerful remote cloud server. We propose an efficient Single Restart with Time Stamps (SRTS) algorithm, which has time complexity $O(n \log n)$. We derive its competitive ratio in comparison with the optimal offline solution. If the unknown processing times are deterministic, the competitive ratio of SRTS is shown to be always constant when the processing times are independent of m, and asymptotically constant in practice when the processing times are dependent on m. We obtain a similar result when the unknown processing times are random.
- Extending the ideas of SRTS, we further propose a heuristic algorithm SRTS-Multiple (SRTS-M), for the case where there are multiple unknown processors, which also has $O(n \log n)$ time complexity.
- To evaluate the average performance of SRTS and SRTS-M, we show using simulation that they provide substantial gains in reducing the makespan over the best known alternatives, for task processing times generated from typical distributions. We further observe that the gains are much more significant for heavy-tailed distributions.

The rest of this paper is organized as follows. In Section II, we present the related work. The system model is given in Section III. The SRTS algorithm is presented in Section IV, and its competitive ratio is derived in Section V. In Section VI we present the SRTS-M algorithm for the case of multiple unknown processors. We discuss simulation results in Section VII and conclude in Section VIII.

II. Related Work

Scheduling independent tasks on parallel processors is a well-studied problem in theoretical computer science, particularly from the perspective of approximation algorithms. In the following we present relevant works from the literature under offline, online, and semi-online settings.

A. Offline and Online Scheduling on Parallel Processors

Even in the simplest *offline* setting, where the processors are *identical*, i.e., for each task the processing times are the same on all processors, the problem is NP-hard [11]. The classical Longest Processing Time (LPT) algorithm forms a list of the tasks in the descending order of their processing times and schedules the next task from the list on whichever processor that becomes idle first. For $m+m'$ identical processors, LPT provides $\left(\frac{4}{3} - \frac{1}{3(m+m')}\right)$-approximation [15]. Other algorithms with various time complexity and approximation ratios are also available in the literature [16], [17]. For the case of non-identical processors, a 2-approximation algorithm was proposed in [18], [19]. Since in our problem the processing times on one processor are not known a priori, none of the above works are applicable.

In the *online* setting, LS lists the tasks in an arbitrary order and schedules the next task on whichever processor that becomes idle first. It provides a $\left(2 - \frac{1}{m+m'}\right)$ competitive ratio for scheduling on $m + m'$ identical processors. LS can be applied to solve our problem, by ignoring the known processing times. However, we will show later that, due to the non-identical processing times among the processors in our problem, a family of deterministic algorithms that include LS has infinite competitive ratio.

Shmoys *et. al.* in [13] considered the general problem of online scheduling of independent tasks on non-identical processors. Using an approach involving multiple rounds of task restarts, they proposed an $O(\log n)$-competitive online algorithm. Similarly to LS, Shmoys' algorithm can be applied to our problem by simply ignoring the known processing times. However, it has been shown in [14] that the average performance of Shmoys' algorithm can suffer due to the multiple rounds of task restarts. We will show in Section VII that even a semi-online improvement of this algorithm can give substantially worse average performance than the proposed SRTS algorithm. Furthermore, by judiciously utilizing the known processing times, SRTS achieves a competitive ratio that is independent of n.

B. Semi-online Scheduling on Parallel Processors

Studies under semi-online settings are comparatively scarce. Even the definition of semi-online scheduling is not unified. In [20]–[22], it refers to the case where only the total processing time of the tasks on each processor is known. In [23], it refers to the case where the processing time of a task is unknown but its communication delay is known. Furthermore, all of these works focus on the special case of identical or uniform processors, so they are not applicable to our problem.

To the best of our knowledge, the semi-online setting most similar to ours is in [14], which may be viewed as having one processor with known processing times (which actually was used to model some fixed usage cost in [14]) and m identical processors with unknown processing times. However, the Greedy-One-Restart (GOR) algorithm proposed in [14] cannot be applied to our problem. While GOR schedules tasks on the m *unknown identical* processors using estimated processing times based on the single known processor, SRTS schedules tasks on the m *known* processors directly using the known processing times. Furthermore, the estimation of the unknown processing times for task restarting requires different methods. It depends on m in GOR, while in SRTS the known processing times are directly used as the estimate. Notably, as a result of these differences, the competitive ratio of SRTS is constant under general conditions, in contrast to $O(\sqrt{m})$ for GOR. Furthermore, we consider the case of multiple unknown processors that are non-identical in the SRTS-M algorithm.

C. Other Related Works

In mobile cloud computing systems [2], [3], where a mobile device enlists the help of a remote processor in a remote cloud, most current research on task offloading is focused on the objective of minimizing energy, e.g., [24]–[26]. In addition, several empirical studies have been conducted on task offloading from a mobile device to remote servers [6]–[10]. Furthermore, the hybrid cloud computing architecture, where tasks are offloaded from a local cluster/cloud to a public cloud, has been considered in [27]–[29]. However, all of these works have system models different from ours, and none of them considers makespan as their design objective. In this work, our focus is to provide a general semi-online solution to the problem of makespan minimization in parallel task scheduling, which may be applied to cloud computing and other practical computing and networked systems.

III. System Model

For clarity of presentation, we initially focus on a heterogeneous system comprised of m identical "known" (or "local") parallel processors indexed by $i \in \mathcal{Q} = \{1, \ldots, m\}$ and a single "unknown" (or "remote") processor indexed by $i = m + 1$. In this work we use the terms "remote processor" and "unknown processor" interchangeably. Given n tasks, indexed by $j \in \mathcal{T} = \{1, \ldots, n\}$, our objective is to minimize the makespan to process them. The tasks are assumed independent and non-preemptive. Even though we initially consider a single unknown processor and propose SRTS, later in Section VI we consider the case of $m' > 1$ unknown processors, for which we propose SRTS-M.

The processing time of task j on processors in $\mathcal{Q}$ is denoted by a_j and is assumed to be known a priori. This may be obtained, for example, by checking the number of instructions per task and the processor speed. The processing time of task j on processor $m+1$ is denoted by u_j and is assumed to be unknown until the task has been executed to completion. This may arise in many scenarios of practical interest. For example, a remote server may be shared and only a fraction of the CPU cycles are allocated to the offloaded tasks. In scenarios where the remote server is dedicated to the offloaded tasks, there may be other uncertain delays in offloading and processing the tasks. We do not assume any relation between u_j and a_j, but it is important to note that our results can serve as a benchmark to evaluate the performance of algorithms that do consider the relation between u_j and a_j.

Note that even though u_j is unknown, it may be deterministic, i.e., it remains the same independent of when task j is processed. For example, this may model the case where the remote CPU cycles allocated to the tasks do not change frequently. However, u_j may also be random, i.e., it depends on when task j is processed. As shown in Section V, this distinction is important in performance analysis, since the proposed SRTS algorithm may cancel and then restart a task at a different time. In this work, we consider both cases.

Let $\mathbf{s}$ denote a schedule and $\mathcal{S}$ denote the set of all possible schedules. The schedule $\mathbf{s}$ decides the placement of a task on one of the local processors $\mathcal{Q}$ and the remote processor $m+1$. Given the set of tasks at time 0, the makespan of a schedule $\mathbf{s}$, denoted by $C_{\max}(\mathbf{s})$, is defined as the time when the processing of the last task is completed. It equals $\max_i\{C_i(\mathbf{s})\}$, where $C_i(\mathbf{s})$ is the completion time of the last task assigned to processor i. We are interested in the following makespan minimization problem $\mathcal{P}$:

$$\underset{\mathbf{s}\in\mathcal{S}}{\text{minimize}} \quad C_{\max}(\mathbf{s}).$$

From Section II, we see that even for the offline version of $\mathcal{P}$, where all parameter values of the tasks are known at time zero, the problem is NP-hard [11]. We are interested in the more complicated semi-online setting, where u_j are not known a priori.

The efficacy of an online algorithm is often measured by its competitive ratio in comparison with an optimal offline algorithm. We use the same measure for semi-online algorithms as well. Let $\{P, \{u_j\}\}$ be a problem instance of $\mathcal{P}$, where $P = \{m, n, \{a_j\}\}$. Let $\mathbf{s}(P, \{u_j\})$ be the schedule given by a semi-online algorithm and $\mathbf{s}^*(P, \{u_j\})$ be the schedule given by an optimal offline algorithm. If u_j are deterministic, then the problem instance $\{P, \{u_j\}\}$ is a set of constants and an optimal offline algorithm outputs the minimum makespan $C_{\max}(\mathbf{s}^*(P, \{u_j\}))$. In this case the semi-online algorithm is said to have a competitive ratio θ if

$$\max_{\forall\{P,\{u_j\}\}} \frac{C_{\max}(\mathbf{s}(P, \{u_j\}))}{C_{\max}(\mathbf{s}^*(P, \{u_j\}))} \le \theta. \tag{1}$$

For the case where u_j are random, we redefine the competitive ratio θ as

$$\max_{\forall P} \frac{\mathbb{E}[C_{\max}(s(P, \{u_j\}))]}{\mathbb{E}[C_{\max}(s^*(P, \{u_j\}))]} \le \theta, \tag{2}$$

where the expectations are taken over the randomness in $\{u_j\}$.

We note that, in the rest of this paper, we implicitly assume that a processor executes one task at a time. However, this is without loss of generality, since sharing the cycles of the processor by those tasks would result in the same completion time.

IV. Single Restart with Time Stamps (SRTS)

In this section we focus on the important case of $m' = 1$. We first present our design consideration for SRTS, then describe the algorithm details, and finally present an illustrative example to explain the working of SRTS.

A. Design Considerations

In this subsection we explain the failure of some existing algorithms for our problem and derive insights into the design of SRTS.

*1) **Failure of algorithms with pre-determined scheduling order:*** We note that the celebrated LS algorithm can be used to solve $\mathcal{P}$ as it does not require the processing times of the tasks on any processor. Also, one can extend the LPT algorithm to solve $\mathcal{P}$ by sorting tasks based on the known a_j values. In the rest of this paper we term this algorithm Semi-Online LPT (SO-LPT). Both LS and SO-LPT belong to the family of algorithms with a *pre-determined scheduling order*, which is formally defined as algorithms that rank the tasks according to some rule and then schedule them one after another in that fixed order. In the following, we study the performance of these algorithm.

In Section II, we have noted that when all processors are identical, LS has a constant competitive ratio. Also, if all processors are identical and the processing times of the tasks are known, then LPT has $\frac{4}{3}$ approximation ratio [15]. Since our problem model has m known identical processors with only one unknown processor, one may expect that there exists some deterministic scheduling algorithm that gives a low competitive ratio. However, in the following theorem, we observe that the family of all algorithms with a per-determined scheduling order are highly ineffective in the worst case. Therefore, we need a more flexible dynamic scheduling approach in our design of SRTS.

Theorem 1. *Any algorithm with pre-determined scheduling order has infinite competitive ratio with respect to $\mathcal{P}$.*

Proof. The proof is given in [30]. □

*2) **Inefficiency of multiple rounds of restarts:*** In Section II, we have noted that Shmoys' online algorithm is the only existing algorithm that can be applied to solve $\mathcal{P}$ with a provable competitive ratio. Shmoys' algorithm initially estimates the unknown processing times of the tasks and then uses any ρ-approximation offline algorithm to schedule them. Tasks that are not completed within the estimated time are cancelled and rescheduled using an increased estimate for the unknown processing times and the same offline algorithm. The procedure is repeated until all tasks are completed. This algorithm has $(4\rho\log n+4\rho\log 2\rho+1)$ competitive ratio [13]. However, its average performance may be unsatisfactory [14], and its competitive ratio still depends on n.

One might consider improving Shmoys' algorithm to a semi-online version to solve $\mathcal{P}$, by incorporating the information about known processing times a_j. We term this improved version *Semi-Online Shmoys* (SO-Shmoys) in the rest of this paper. In SO-Shmoys, we use a_j as the initial estimate of the unknown processing time u_j, and LPT as the offline component algorithm. For each iteration, the estimated processing time is doubled. In iteration k, since LPT is applied to an offline problem where the processing time of a task is a_j on the first m processors and $2^k a_j$ on processor $(m+1)$, it yields 2 approximation [31] for all k. Thus, overall SO-Shmoys remains $O(\log n)$-competitive, and its average performance is improved. However, as shown in Section VII, we observe that SO-Shmoys does not perform better than SO-LPT in terms of average performance. This is due to the multiple rounds of task restarts, each penalizing the makespan, since the time already spent on processing a cancelled task is wasted.

Therefore, in SRTS we use only a single round of task restarts. Cancelling a task with large u_j on processor $m+1$ may allow some tasks that have smaller u_j values to be scheduled on that processor. At the same time, we avoid the wastage of time in cancelling a task more than once. As shown in Sections V and VII, our new design achieves both a small competitive ratio and superior average performance. A detailed description of SRTS is given below.

B. SRTS Algorithm Description

SRTS is comprised of two iterations. In the initial iteration, it first uses a_j as the estimate for the processing time of task j on the unknown processor $m+1$. It forms a list according to the ascending order of a_j. When processor $m+1$ becomes idle, it schedules the next available task from the *end* of the list. If the task is not completed within duration a_j, it cancels the task and sets it aside. Whenever a processor in the known processor set $\mathcal{Q}$ becomes idle, it schedules the next available task from the *start* of the list. We note that the above scheduling order of tasks is advantageous for tasks that incur large a_j and small u_j values.

After going through all tasks in the first iteration above, those tasks that are cancelled are again sorted, and a list is formed in the ascending order of a_j. In the second iteration, this list is scheduled using the same procedure as above, but this time we do not cancel a task, unless it is simultaneously scheduled on two processors. Note that in both iterations some tasks may be scheduled on both processor $m+1$ and some processor in $\mathcal{Q}$. In such a case we cancel the task on one processor if it is either *completed or cancelled* on another processor first.

SRTS can be readily implemented in practice by a scheduler in a local device or a local cluster. For example, this can be achieved by assigning time stamps to the tasks that are offloaded. A remote processor looks at the time stamp of a task to determine when to discard it. The scheduler at the local processor decides to restart an offloaded task if it does not receive an acknowledgement or the output of the task within the duration specified by time stamp.

The details of the algorithm are presented in Algorithm 1, where $l=1$ or 2 indicates the iteration number. We note that SRTS runs in $O(n\log n)$ time due to the need for sorting n tasks. We use $\mathrm{s}^{\mathrm{SRTS}}$ to denote the resultant schedule.

C. Illustrative Example

In the following, we explain the working of SRTS using the following family of problem instances: $u_1=\alpha>1$,

Algorithm 1: Single Restart with Time Stamps (SRTS)

```
1: $\mathcal{T}^{(1)} = \mathcal{T}$
2: **for** $l = 1$ to $2$ **do**
3:   Sort $\mathcal{T}^{(l)}$ in the ascending order of $a_j$. WLOG,
     re-index tasks such that $a_1 \leq a_2 \leq \ldots \leq a_{|\mathcal{T}^{(l)}|}$.
4:   $j_1 = |\mathcal{T}^{(l)}|$, $j_0 = 0$
5:   Start processing task $j_1$ on processor $m+1$
6:   **if** $l = 1$ **then**
7:     Cancel task $j_1$ if its execution time
       exceeds $a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
8:   **end if**
9:   **for** $k = 1$ to $\min\{m, |\mathcal{T}^{(l)}|\}$ **do**
10:    $j_0 = j_0 + 1$
11:    Start processing task $j_0$ on processor $k$.
12:  **end for**
13:  **while** $\mathcal{T}^{(l)} \neq \emptyset$ **do**
14:    Wait until next event $E$ occurs
15:    **if** $E =$ a processor $\hat{i} \in \mathcal{Q}$ becomes idle **then**
16:      Let task $j$ be the last task completed on $\hat{i}$
17:      Cancel task $j$ if it is scheduled on processor
         $m+1$
18:      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j\}$
19:      $j_0 = j_0 + 1$
20:      If task $j_0$ is not completed or cancelled yet,
         schedule it on processor $\hat{i}$
21:    **else if** $E =$ processor $m+1$ becomes idle **then**
22:      Cancel task $j_1$ if it is scheduled on some
         processor from $\mathcal{Q}$
23:      $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j_1\}$
24:      $j_1 = j_1 - 1$
25:      If task $j_1$ is not completed yet, schedule it on
         processor $m+1$
26:      **if** $l = 1$ **then**
27:        Cancel task $j_1$ if its execution time exceeds
           $a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
28:      **end if**
29:    **end if**
30:  **end while**
31: **end for**
```

$u_j = 1, \forall j \in \{2, \ldots, n\}$, and $a_j = 1, \forall j$. For simplicity of illustration, we further assume that n is a multiple of $m+1$. Since $a_j = 1, \forall j$, SRTS do not differentiate the tasks. Let us consider the worst case scenario, where task 1 is scheduled on processor $m+1$ in the first iteration of SRTS. Note that in the first iteration of SRTS, any task scheduled on processor $m+1$ is processed for a duration of $\min\{a_j, u_j\}$, which is equal to 1 for all tasks. Therefore, task 1 will be cancelled after a duration of 1. At the end of the first iteration, $n - \frac{n}{m+1}$ tasks will be completed on the processors in $\mathcal{Q}$ and $\frac{n}{m+1} - 1$ of them will be completed on processor $m+1$, with task 1 being cancelled. Then, in the second iteration of SRTS, task 1 will be completed on some processor in $\mathcal{Q}$.

In the first iteration, the $n - \frac{n}{m+1}$ tasks on the processors in $\mathcal{Q}$ require a duration of $\frac{1}{m}\left(n - \frac{n}{m+1}\right) = \frac{n}{m+1}$ on each processor. On processor $m+1$, the duration is also $\frac{n}{m+1}$. The duration of the second iteration is 1. Therefore, the makespan of SRTS for these problem instances is $\frac{n}{m+1} + 1$, which is independent of the unknown processing time α. Note that, since the processing time of any task on any processor is at least 1, the offline optimal makespan cannot be less than $n/(m+1)$. This example illustrates that, by restarting a task that has larger u_j value, SRTS can effectively limit the impact of that task on the makespan.

V. Competitive Ratio Analysis

In this section, we first consider the case of deterministic u_j and derive a competitive ratio θ for SRTS. Then, we extend the result to the case of random u_j, showing that the same competitive ratio θ holds with only minor modification.

A. Deterministic u_j

In each iteration l of Algorithm 1, where $l = 1$ or 2, we consider the following intermediate outcome of SRTS that will be used extensively in our analysis and proofs. Let $\mathbf{s}_l$ denote the intermediate schedule in iteration l obtained by breaking the loop in Line 13 of Algorithm 1 as soon as j_0 becomes equal to j_1. We note that $\mathbf{s}_l$ is a schedule over the set $\mathcal{T}^{(l)}$, and all the tasks from $\mathcal{T}^{(l)}$ will be scheduled at least once under $\mathbf{s}_l$. To understand this, in the while loop from Line 13 of Algorithm 1, when $j_0 = j_1 - 1$, all the $|\mathcal{T}^{(l)}|$ tasks should have been scheduled on some processor. Now, any more iterations in the while loop will only result in scheduling a task that is already scheduled on processor $m+1$ onto some processor in $\mathcal{Q}$, or vice-versa. Since under $\mathbf{s}_l$ the while loop breaks when $j_0 = j_1$, there will be only one task that is scheduled on both processor $m+1$ and some processor in $\mathcal{Q}$. This will be the last task scheduled by $\mathbf{s}_l$ in iteration l, and we denote it by $q^{(l)} = j_0 = j_1$.

We refer to the time to process the set of tasks $\mathcal{T}^{(l)}$ in iteration l as the *schedule length* of this iteration, denoted by $C^{(l)}_{\text{max}}$. In the rest of this paper, to differentiate the terms with respect to $\mathbf{s}^{\text{SRTS}}$ and $\mathbf{s}_l$, we append onto them the labels of $(\mathbf{s}^{\text{SRTS}})$ and $(\mathbf{s}_l)$, respectively. We note that in iteration l, the schedule produced by $\mathbf{s}^{\text{SRTS}}$ improves on $\mathbf{s}_l$. To see this, observe that $\mathbf{s}_l$ stops scheduling when $j_0 = j_1$. The step $j_0 = j_1$ also occurs under $\mathbf{s}^{\text{SRTS}}$ in both iterations. However, $\mathbf{s}^{\text{SRTS}}$ may not stop at this step. If processor $m+1$ is faster and completes task $q^{(l)} - k$ first, where $k \in \{0, 1, \ldots, \min\{q^{(l)}, m\} - 1\}$, then $\mathbf{s}^{\text{SRTS}}$ schedules task $q^{(l)} - k - 1$, if it is not completed yet, onto processor $m+1$. This will result in a schedule length no longer than that given by $\mathbf{s}_l$, i.e., $C^{(l)}_{\text{max}}(\mathbf{s}^{\text{SRTS}}) \leq C^{(l)}_{\text{max}}(\mathbf{s}_l)$.

Note that, in the analysis and proofs that follow, we do not explicitly mention problem instance $\{P, \{u_j\}\}$, as the results are valid over all possible problem instances. Also, we simply use $\mathbf{s}^*$ to denote an optimal offline schedule and C^*_{max} to denote the offline optimal makespan.

In Lemma 1, using load balancing arguments we establish a relation between $C^{(l)}_{\text{max}}(\mathbf{s}_l)$ and the known processing times a_j.

Lemma 1.

$$mC^{(l)}_{max}(\mathbf{s}_l) \leq \sum_{j \in \mathcal{T}^{(l)}} a_j + (m-1)a_{q^{(l)}},$$

where task $q^{(l)}$ is the last task scheduled, in iteration l, under schedule $\mathbf{s}_l$.

Proof. The proof is given in Appendix A. □

In the first iteration of SRTS, task j scheduled on processor $m+1$ is processed for duration $\min\{u_j, a_j\}$, since it is cancelled if its processing duration exceeds a_j. We use this fact and Lemma 1 to derive an upper bound for $C^{(1)}_{\max}(\mathbf{s}^{\text{SRTS}})$, which is given in Lemma 2.

Lemma 2.

$$C^{(1)}_{max}(\mathbf{s}^{SRTS}) \leq \min\left\{2 + \frac{\beta_{max} - 2}{m+1}, m+1\right\} C^*_{max},$$

where $\beta_{max} = \max_j \frac{a_j}{u_j}$.

Proof. Due to space limitation, the proof is given in [30]. □

A task j scheduled in the second iteration has the property $u_j > a_j$. Using this fact along with Lemma 1, we arrive at Lemma 3.

Lemma 3. $C^{(2)}_{max}(\mathbf{s}^{SRTS}) \leq 2C^*_{max}$

Proof. Due to space limitation, the proof is given in [30]. □

Noting that $C_{\max}(\mathbf{s}^{\text{SRTS}}) = C^{(1)}_{\max}(\mathbf{s}^{\text{SRTS}}) + C^{(2)}_{\max}(\mathbf{s}^{\text{SRTS}})$, the following theorem immediately follows from Lemmas 2 and 3:

Theorem 2. *For deterministic u_j, SRTS is θ-competitive for $\mathcal{P}$, where*

$$\theta = \min\left\{4 + \frac{\beta_{max} - 2}{m+1}, m+3\right\}. \tag{3}$$

From Theorem 2 it can be observed that SRTS yields a competitive ratio with some interesting features. First, unlike in the case of SO-Shmoys, θ is independent of n. This is important, since the number of tasks in common applications such as cloud computing can be large.

Second, if $\beta_{\max}$ is independent of m, then a simple upper bound for θ in terms of $\beta_{\max}$ can be obtained by solving for m in the following equation:

$$4 + \frac{\beta_{\max} - 2}{m+1} = m + 3.$$

The solution is given by $m = \sqrt{\beta_{\max} - 1}$. Substituting this value in (3) and noting that $m \geq 1$, we obtain

$$\theta \leq \begin{cases} 4, & 0 < \beta_{\max} < 2 \\ \sqrt{\beta_{\max} - 1} + 3, & \beta_{\max} \geq 2. \end{cases}$$

Therefore, in this case SRTS has constant competitive ratio independent of m.

Third, consider the case where $\beta_{\max}$ is a function of m. As an example, this may happen if we assume that the capacity of the remote processor is always at a similar level as the combined capacity of all m local processors. From (3), we observe that as long as $\beta_{\max}$ is $O(m)$, θ is $O(1)$. In other words, if the unknown processing speed is $O(m)$ times the processing speed of each known processor, SRTS has asymptotically constant competitive ratio. Note that in most practical parallel computing systems, the speed difference between the unknown (e.g., cloud) processor and a known (e.g., local) processor is not excessive. Therefore, in this case we expect SRTS to have asymptotically constant competitive ratio in general.

B. Worst Case Bound for Random u_j

In the case of random u_j, if a task is restarted on processor $m+1$ under SRTS, it acquires a different processing time. Therefore, $C^*_{\max}$ and $C_{\max}(\mathbf{s}^{\text{SRTS}})$ are not directly comparable. Nevertheless, we may compare their expected values over the random realizations of u_j. Here, we observe that Theorem 2 can be generalized to the competitive ratio definition in (2).

Theorem 3. *For random u_j, SRTS has the following upper bound for the expected makespan ratio:*

$$\frac{\mathbb{E}[C_{max}(\mathbf{s}(P, \{u_j\}))]}{\mathbb{E}[C_{max}(\mathbf{s}^*(P, \{u_j\}))]} \leq \min\left\{4 + \frac{\max_j(\frac{a_j}{\nu_{min}}) - 2}{m+1}, m+3\right\},$$

where $\nu_{min} > 0$ is the minimum value in the sample space from which u_j are drawn.

Proof. The proof is given in [30]. □

VI. Extension to Multiple Unknown Processors

In this section, we consider the problem where multiple remote processors with unknown processing times are available for offloading the computational tasks. We consider the general case where the remote processors are non-identical and index them by $i \in \mathcal{Q}' = \{m+1, \ldots, m+m'\}$. Recall that Shmoys' algorithm can be applied to this case and is $O(\log n)$-competitive if the processing times are deterministic. However, as noted before it has very poor average performance. Instead, learning from the proven ideas of SRTS, we propose a heuristic SRTS-Multiple (SRTS-M) algorithm to solve this problem.

Similar to SRTS, SRTS-M also has two iterations. The tasks are listed in the ascending order of a_j values. Without loss of generality, consider $a_1 \leq a_2 \leq \ldots \leq a_n$. In the first iteration of SRTS-M, whenever a local processor becomes idle, it is given a task from the *start* of the list. Similarly, whenever a remote processor becomes idle it is given a task from the *end* of the list. A task j_1 that is scheduled on a remote processor is cancelled in the first iteration if its processing on the remote processors exceeds the estimation time $\sum_{j=j_1}^{n} a_j$. The rationale behind this choice of the estimation times is the following. Consider a hypothetical powerful single remote processor in place of the set of remote processors, and we use SRTS to schedule the tasks. In this case, in the first iteration of SRTS, the time that any offloaded task j_1 is completed or cancelled is upper bounded by $\sum_{j=j_1}^{n} a_j$. We note that our choice of estimation time $\sum_{j=j_1}^{n} a_j$ for any offloaded task j_1 in SRTS-M is greater than or equal to the estimation time a_{j_1} used in SRTS. This higher estimation time in SRTS-M potentially avoids unnecessary restarts on multiple remote processors.

The details of SRTS-M are presented in Algorithm 2. Similarly to SRTS, SRTS-M runs in $O(n \log n)$ time and can be readily implemented in practice by a local scheduler. However, it is challenging to derive its competitive ratio, because restarting an offloaded task on an unknown processor does not reveal any information about its processing time on another unknown processor, thereby making it difficult to derive an upper bound expression for the makespan. Instead, in Section VII, we show using simulation that it significantly out performs the best existing alternatives.

Algorithm 2: SRTS-M

```
1:  $\mathcal{T}^{(1)} = \mathcal{T}$
2:  for $l = 1$ to $2$ do
3:      Sort $\mathcal{T}^{(l)}$ in the ascending order of $a_j$. WLOG,
        re-index tasks such that $a_1 \leq a_2 \leq \ldots \leq a_{|\mathcal{T}^{(l)}|}$.
4:      $j_1 = |\mathcal{T}^{(l)}| + 1$, $j_0 = 0$
5:      for $k = m + 1$ to $m + \min\{m', |\mathcal{T}^{(l)}|\}$ do
6:          $j_1 = j_1 - 1$
7:          Start processing task $j_1$ on processor $k$
8:          if $l = 1$ then
9:              Cancel task $j_1$ if its execution time
                exceeds $\sum_{j=j_1}^{n} a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
10:         end if
11:     end for
12:     for $k = 1$ to $\min\{m, |\mathcal{T}^{(l)}|\}$ do
13:         $j_0 = j_0 + 1$
14:         Start processing task $j_0$ on processor $k$.
15:     end for
16:     while $\mathcal{T}^{(l)} \neq \emptyset$ do
17:         Wait until next event $E$ occurs
18:         if $E =$ a processor $\hat{i} \in \mathcal{Q}$ becomes idle then
19:             Let task $j$ be the last task completed on $\hat{i}$
20:             Cancel task $j$ if it is scheduled on some
                processor from $\mathcal{Q}'$
21:             $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j\}$
22:             $j_0 = j_0 + 1$
23:             If task $j_0$ is not completed or cancelled yet,
                schedule it on processor $\hat{i}$
24:         else if $E =$a processor $\hat{i} \in \mathcal{Q}'$ becomes idle then
25:             Cancel task $j_1$ if it is scheduled on some
                processor from $\mathcal{Q}$
26:             $\mathcal{T}^{(l)} = \mathcal{T}^{(l)} \backslash \{j_1\}$
27:             $j_1 = j_1 - 1$
28:             If task $j_1$ is not completed yet, schedule it on
                processor $\hat{i}$
29:             if $l = 1$ then
30:                 Cancel task $j_1$ if its execution time exceeds
                    $\sum_{j=j_1}^{n} a_{j_1}$ and include it in $\mathcal{T}^{(l+1)}$
31:             end if
32:         end if
33:     end while
34: end for
```

VII. Evaluation of Average Performance

In addition to the competitive ratios derived for SRTS in Section V, we are interested in studying the average performance of SRTS and SRTS-M over general parameter values. Toward this end, we conduct simulation in MATLAB for evaluation and comparison with several well-known alternatives.

A. Single Remote Processor

We compare SRTS with LS, SO-LPT, and SO-Shmoys. In addition, we also consider a *Semi-Online Shortest Processing Time* (SO-SPT) algorithm, which is the same as SO-LPT except that the known process times a_j are listed in ascending order. For Figures 1 and 2, a_j and u_j are generated independently from an exponential distribution. We set the number of tasks $n = 1500$. For each data point, we average the makespan over $10,000$ runs, combining 100 realizations each for $\{a_j\}$ and $\{u_j\}$. In Figure 1, we set $m = 10$, $E[a_j] = 60$, and vary $E[u_j]$. In Figure 2, we set $E[a_j] = 60$, $E[u_j] = 6$, and vary m. We choose $E[a_j]$ larger than $E[u_j]$ to reflect the practical scenario where the remote server is often faster than the local processors. We observe that SRTS outperforms all other algorithms. It provides a makespan reduction up to 30% compared with the best alternatives of SO-LPT and SO-Shmoys.

Similar performance trends have been observed when we use other distributions. In general, the performance advantage of SRTS is more pronounced when the distribution of a_j and u_j has a heavier tail. This is because a heavier tail implies more chances for some extremely long processing times, which can clog an unknown processor in algorithms with deterministic scheduling order, such as LS and SO-LPT, and lead to high inefficiency in algorithms with multiple restarts and no simultaneous processing, such as SO-Shmoys. This is illustrated in Figure 3, where we generate $\{a_j\}$ and $\{u_j\}$ using the Pareto distribution. The Pareto scale parameters of $\{a_j\}$ and $\{u_j\}$ are given by the Pareto tail index multiplied by 60 and 6, respectively. We note that as the Pareto tail index parameter increases, the *heaviness* of the tail decreases.

B. Multiple Remote Processors

In Figure 4, we compare the average performance of SRTS-M with the alternate algorithms mentioned in Section VII-A. The processing times a_j and u_j are generated independently from exponential distributions. The default parameters are $m = 4$, $\mathbb{E}[a_j] = 60$, and $\mathbb{E}[u_j] = 60$. We observe that SRTS-M provides $20 - 30\%$ reduction in the average makespan over a wide range of m' values.

For task processing times generated using *heavy-tailed* distributions as in the previous section, we observe that SRTS-M significantly reduces the makespan when compared with the alternatives. This is illustrated in Figure 5. Finally, we note that the performance of SO-Shmoys degrades significantly with multiple unknown processors because of the multiple rounds

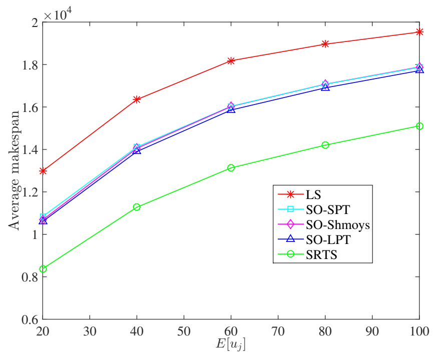

Fig. 1. Effect of varying $\mathbb{E}[u_j]$. Single remote processor.

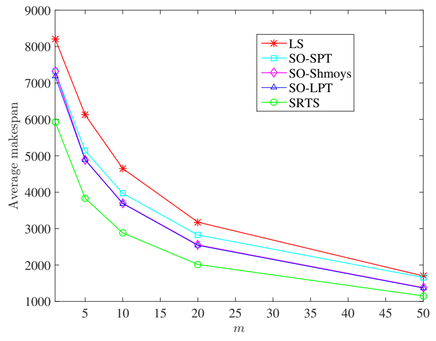

Fig. 2. Effect of the number of local processors. Single remote processor.

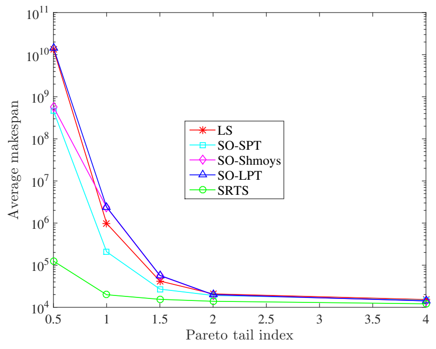

Fig. 3. Effect of the tail of distribution, for $m = 4$. Single remote processor.

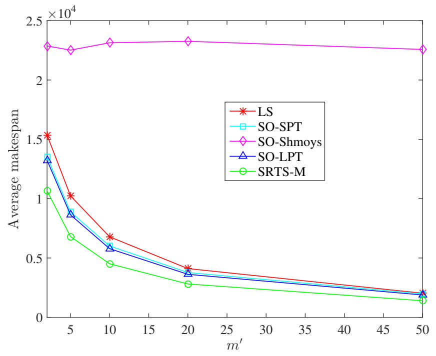

Fig. 4. Effect of the number of remote processors.

of restarts on all the unknown processors.

VIII. Conclusion

We have proposed SRTS and SRTS-M algorithms for semi-online scheduling of n tasks on m identical known processors and one or multiple unknown processors, with an aim to reduce the makespan of processing all tasks. If the unknown task processing times are deterministic, the competitive ratio of SRTS is shown to be always constant when the processing times are independent of m, and asymptotically constant in practice when the processing times are dependent on m. We derive a similar result for the case where the unknown task processing times are random. Furthermore, our simulation results show that SRTS and SRTS-M provide substantial performance improvement over existing alternatives in terms of the average makespan, and the performance improvement is more pronounced if the task processing times follow heavy-tailed distributions.

IX. Appendix

A. Proof of Lemma 1

In iteration l, let $C_i^{(l)}$ denote the schedule length, and $\mathcal{T}_i^{(l)}$ denote the set of tasks scheduled on processor i. We note that, in the definition of $\mathcal{T}_i^{(l)}$, when a task is scheduled on two processors simultaneously, it will be counted only on the processor where it is completed or cancelled first. Recall that under schedule $\mathbf{s}_l$, $q^{(l)}$ is the last task scheduled on processor $m+1$ and some processor, say $\hat{i}$, from $\mathcal{Q}$. Let $C_{\max}^{(l)}(\mathbf{s}_l) = C_{\bar{i}}^{(l)}(\mathbf{s}_l)$ for some processor $\bar{i} \in \mathcal{Q}\cup\{m+1\}$. Now, we consider the following cases.

Case 1: $\bar{i} = m+1$. For this case, task $q^{(l)}$ is scheduled both on processor $m+1$ and processor $\hat{i}$, but completed or cancelled first on processor $m+1$. Therefore, $C_{\max}^{(l)}(\mathbf{s}_l)$ should be smaller than the sum of the processing times of tasks scheduled on processor $\hat{i}$ plus $a_{q^{(l)}}$, i.e.,

$$C_{\max}^{(l)}(\mathbf{s}_l) \le \sum_{j\in\mathcal{T}_{\hat{i}}^{(l)}(\mathbf{s}_l)} a_j + a_{q^{(l)}}. \tag{4}$$

Also, at time $\sum_{j\in\mathcal{T}_{\hat{i}}^{(l)}(\mathbf{s}_l)} a_j$, all the processors in $\mathcal{Q}\backslash\{\hat{i}\}$ should be busy executing some task, since otherwise the task $q^{(l)}$ would have been scheduled on that processor which is idle before this time. Therefore,

$$\begin{aligned} \sum_{j\in\mathcal{T}_{\hat{i}}^{(l)}(\mathbf{s}_l)} a_j &\le \sum_{j\in\mathcal{T}_{i}^{(l)}(\mathbf{s}_l)} a_j, \forall i \in \mathcal{Q}\backslash\{\hat{i}\} \\ \Rightarrow C_{\max}^{(l)}(\mathbf{s}_l) &\le \sum_{j\in\mathcal{T}_{i}^{(l)}(\mathbf{s}_l)} a_j + a_{q^{(l)}}, \forall i \in \mathcal{Q}\backslash\{\hat{i}\}. \end{aligned} \tag{5}$$

In the second inequality above, we have used (4). Since task $q^{(l)}$ is completed or cancelled first on processor $m+1$, $q^{(l)} \in \mathcal{T}_{m+1}^{(l)}(\mathbf{s}_l)$. Note that in Algorithm 1, the tasks are listed in the ascending order of a_j and then the tasks from the start of the list are scheduled on processors in $\mathcal{Q}$. This implies $\cup_{i\in\mathcal{Q}}\mathcal{T}_i^{(l)}(\mathbf{s}_l) = \{1,\ldots,q^{(l)}-1\} \subseteq \mathcal{T}^{(l)}$. We use these

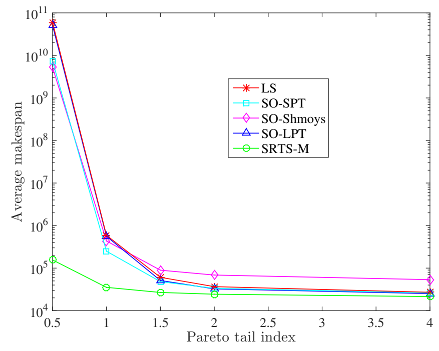

Fig. 5. Effect of the tail of distribution, for $m = 4$, and $m' = 4$.

observations and summing (4) and (5) to obtain the intended result.

Case 2: $\bar{i} = \hat{i}$. The proof of this case is similar to **Case 1** and is given in [30].

Case 3: $\bar{i} \notin \{\hat{i}, m+1\}$. We claim that for this case task $q^{(l)}$ is completed or cancelled first on processor $m+1$. Note that processors from $\mathcal{Q}$ are identical, and tasks are sorted in the ascending order of a_j and re-indexed such that $a_1 \leq a_2 \leq \ldots \leq a_{q^{(l)}}$. This implies that task $q^{(l)}$ has the largest processing time among the tasks scheduled on processors in $\mathcal{Q}$, and it has the latest starting time. If task $q^{(l)}$ were completed on processor $\hat{i}$, then $C^{(l)}_{\max}(\mathbf{s}_l) = C^{(l)}_{\hat{i}}(\mathbf{s}_l)$, which would be a contradiction for this case since $\bar{i} \neq \hat{i}$.

Now, completing task $q^{(l)}$ on processor $\hat{i}$ would have increased the schedule length. Further, task $q^{(l)}$ is scheduled on processor $\hat{i}$ because at the time when processor $\hat{i}$ becomes idle and $q^{(l)}$ is the next task to be scheduled, all other processors are busy executing some task. The above two observations imply that scheduling and completing task $q^{(l)}$ on any of the processors in $\mathcal{Q}$ would have increased the schedule length. This results in similar inequalities as in Case 1, and using the same manipulation we can obtain the intended result.

References

[1] J. Sgall, *Online Algorithms: The State of the Art*. Berlin, Heidelberg: Springer, 1998, ch. On-line scheduling, pp. 196–231.

[2] M. R. Rahimi, J. Ren, C. H. Liu, A. V. Vasilakos, and N. Venkatasubramanian, "Mobile cloud computing: A survey, state of art and future directions," *Mob. Netw. Appl.*, vol. 19, no. 2, pp. 133–143, 2014.

[3] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013.

[4] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing - a key technology towards 5g," European Telecommunications Standards Institute (ETSI) White Paper, 2015.

[5] B. Liang, *"Mobile edge computing," in Key Technologies for 5G Wireless Systems*, V. W. S. Wong, R. Schober, D. W. K. Ng, and L.-C. Wang, Eds. Cambridge University Press, 2017.

[6] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009.

[7] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 87–92.

[8] M. Conti and M. Kumar, "Opportunities in opportunistic computing," *Computer*, vol. 43, no. 1, pp. 42–50, Jan. 2010.

[9] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '12. New York, NY, USA: ACM, 2012, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/2248371.2248394

[10] M. Pitkänen, T. Kärkkäinen, J. Ott, M. Conti, A. Passarella, S. Giordano, D. Puccinelli, F. Legendre, S. Trifunovic, K. Hummel, M. May, N. Hegde, and T. Spyropoulos, "Scampi: Service platform for social aware mobile and pervasive computing," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 503–508, Sep. 2012.

[11] M. Drozdowski, *Scheduling for Parallel Processing*. Springer Publishing Company, 2009.

[12] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, pp. 1563–1541, 1966.

[13] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line," *SIAM J. Comput.*, vol. 24, no. 6, pp. 1313–1331, Dec. 1995.

[14] J. P. Champati and B. Liang, "One-restart algorithm for scheduling and offloading in a hybrid cloud," in *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*, Jun. 2015.

[15] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[16] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, no. 2, pp. 287–326, 1979.

[17] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[18] C. N. Potts, "Analysis of a linear programming heuristic for scheduling unrelated parallel machines," *Discrete Applied Mathematics*, vol. 10, no. 2, pp. 155–164, 1985.

[19] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Math. Program.*, vol. 46, no. 3, pp. 259–271, Feb. 1990.

[20] H. Kellerer, V. Kotov, M. G. Speranza, and Z. Tuza, "Semi on-line algorithms for the partition problem," *Operations Research Letters*, vol. 21, no. 5, pp. 235 – 242, 1997.

[21] T. E. Cheng, H. Kellerer, and V. Kotov, "Semi-on-line multiprocessor scheduling with given total processing time," *Theoretical Computer Science*, vol. 337, no. 13, pp. 134 – 146, 2005.

[22] S. Albers and M. Hellwig, "Semi-online scheduling revisited," *Theoretical Computer Science*, vol. 443, no. 0, pp. 1 – 9, 2012.

[23] J. P. Champati and B. Liang, "Semi-online algorithms for computational task offloading with communication delay," to appear in the *IEEE Transactions on Parallel and Distributed Systems*.

[24] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010, pp. 49–62.

[25] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, 2012, pp. 945–953.

[26] J. Champati and B. Liang, "Energy compensated cloud assistance in mobile cloud computing," in *Proc. IEEE INFOCOM Workshop on Mobile Cloud Computing*, April 2014.

[27] M. Rahman, X. Li, and H. N. Palit, "Hybrid heuristic for scheduling data analytics workflow applications in hybrid cloud environment," in *Proc. IEEE IPDPS Workshops*, 2011, pp. 966–974.

[28] X. Qiu, W. L. Yeow, C. Wu, and F. C. M. Lau, "Cost-minimizing preemptive scheduling of mapreduce workloads on hybrid clouds," in *Proc. IEEE IWQoS*, 2013, pp. 213–219.

[29] M. Shifrin, R. Atar, and I. Cidon, "Optimal scheduling in the hybrid-cloud," in *Proc. IFIP/IEEE International Symposium on Integrated Network Management*, 2013, pp. 51–59.

[30] J. P. Champati and B. Liang, "Single restart with time stamps for computational offloading in a semi-online setting (technical report)," 2017. [Online]. Available: http://www.comm.utoronto.ca/%7eliang/publications/techreport/INFOCOM2017TechRepSRTS.pdf

[31] T. Gonzalez, O. H. Ibarra, and S. Sahni, "Bounds for LPT Schedules on Uniform Processors," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 155–166, 1977.